Consider the following program in supercombinator form:

```
$from n = n : $from ($succ n)
$succ n = n + 1
$prog = $from ($succ 0)
```

We shall give the code along with snapshots of execution:

```
Startup code
------------
1. pushglobal $prog

-------------
        ---------$prog
-------------

2. eval
-------------
        ---------$prog
-------------
saves the PC and the SP and jumps to $prog code

3. print

comes back to this point after the eval in 2. is complete print
prints the eval-ed object. We shall discuss the details of print
later.

4. halt

halts the machine


$prog code
----------

1. pushint 0

-------------
        ---------$prog
-------------
        ---------0
-------------
```

2. pushglobal succ

```
-------------
        ---------$prog
-------------
        ---------0
-------------
        ---------succ
-------------
```

3. mkap

```
-------------
        ---------$prog
-------------
        -------------@
-------------       / \
                  succ  0
```

4. pushglobal $from

```
-------------
        ---------$prog
-------------
        -------------@
-------------       / \
        ------  $succ  0
------------- \
             $from
```

5. mkap

```
-------------
        ---------$prog
-------------
        -------------@
-------------       / \
                $from  @
                      / \
                 $succ   0
```

6. update 1

```
-------------
        -------------@
-------------       / \
                $from  @
                      / \
                 $succ   0
```

7. unwind

```
-------------
        -------------@      travels down the spine, identifies
-------------       / \     the tip, manipulates pointers
        -----#  $from  @#   so that they start pointing to the
-------------         / \   arguments, jumps to the code of
                 $succ   0  $from.
```

$from code
----

1. push 0

```
-------------
        -------------@
-------------       / \
        -----#  $from  @#
-------------         / \
        -----#     $succ   0
-------------
```

2. pushglobal $succ
3. mkap
4. pushglobal $from
5. mkap

```
-------------
        -------------@
-------------       / \
        -----#  $from  @#
-------------         / \
        -----*   $succ   0
```

```
 -------------
                          * @
                           / \
                        $from @
                             / \
                         $succ  #

6. push 1

 -------------
         -------------@
 -------------       / \
         -----#  $from  @#
 -------------         / \
         -----*   $succ   0
 -------------
         -----#
 -------------             *@
                           / \
                        $from @
                             / \
                         $succ  #
7. cons

 -------------
         -------------@
 -------------       / \
         -----#  $from  @#
 -------------         / \
         -----^   $succ   0
 -------------
                     ^:
                     / \     *@
                    #   *   / \
                        $from @
                             / \
                         $succ  #


8. update 2
9. pop 1
                       @#
 -------------         / \
```

```
        -----^   $succ   0
-------------
                    ^:
                    / \     *@
                   #   *    / \
                       $from @
                            / \
                        $succ  #
```

10. unwind — unwind finds that the tip of the redex is in WHNF and therefore returns.

THE GENERAL COMPILATION SCHEME

The startup code is always:

```
pushglobal $prog
eval
print
halt
```

The code for a supercombinator is given by the F compilation function:

```
F [$f x1 x2 ... xn = E]
```

To define F, note that the situation just before entering a supercombinator body is:

```
    ------------------
 ^           -----------------------@
 |  ------------------             / \
 |           ---------argn        @  argn
d+1 ------------------           / \
 |           ---------arg2      @  arg2
 |  ------------------         / \
 v           ---------arg1    @  arg1
    ------------------       /
                            $f
```

The distance from the top of the stack to the root of the redex is called the context and is length is denoted as d+1. The initial value of d is n and changes as the execution of the supercombinator body proceeds.

The distance between argi and the root of the redex is given by an environment function rho, where rho argi = n+1-i

Therefore the distance of an argument i from the top of the stack is d - (rho argi). This is how we keep track of the position of an argument from the top of the stack. Once again rho is constant for a supercombinator, but d has to be tracked as compilation proceeds.

```
F[$f x1 x2 ... xn = E] =
 globstart $f, n:
      R[E] (x1=n, x2=n-1, ...xn=1)
```

```
R[E] rho d = C[E] rho d; update d+1; pop d; unwind
```

The definition of C is by cases

```
C[E] rho d = case E of
               E is an integer i -> pushint i
               E is a sc $g -> pushglobal $g
               E is an argument x -> push (d - rho x)
               E is an application e1 e2 -> C[e2] rho d;
                                            C[e1] rho (d+1);
                                            mkap
               E is a let expression let x = Ex in Eb ->
                   C[Ex] rho d;
                   C[Eb] rho[x/d+1] (d+1);
                   slide 1
                   -- first compute Ex on top of the stack. Evaluate Eb
                   -- in the updated environment. Push down (slide) the
                   -- value of Eb overwriting the value of Ex

               E is a letrec expression letrec D = (x1 = E1
                                                    x2 = E2
                                                    xn = En)
                                        in E
                   ->
                   Cletrec[D = (x1=E1 x2=E2 xn=En)] rho' d';
                   C[Eb] rho' d';
                   slide (d'-d)
                       where (rho', d') = Xr[D] rho d
                             Xr[D] rho d = (rho[x1/d+1,...xn/d+n], d+n)
```

```
            Cletrec[D] rho d = alloc n;
                               C[E1] rho d; update n;
                               ...
                               C[En] rho d; update 1;
    Xr creates an environment in which x1..xn refer to
    the n locations above the top of the stack and
    lengthens the current context by n

    Cletrec actually allocates n locations and creates
    pointers pointing to them from the top of stack.

    Then it constructs the graphs corresponding to E1
    to En, and overwrites the allocated locations with
    these graphs.

    Finally Eb is evaluated in the letrec  environment
    and slid into position


E is cons -> pushglobal $cons
        where $cons code is
            cons
            update 1
            unwind / return

    cons is the g-machine instruction pops the top two
    elements of the stack and makes a cons out of them
    The root of the redex is updated and since unwind will
    find : which is in WHNF. we can use a return instead.
E is +   -> pushglobal $+
        where $cons code is
            push 1
            eval
            push 1
            eval
            add
            update 3
            pop 2
            unwind / return

E is head ->  pushglobal $head
        where $head code is
            eval     - expose the cons cell
```

```
            head     - pick the head
            eval     - evaluate the head before updation
            update 1   else there will be duplicate
                       evaluation SPJ - 12.4
            unwind     Show that a return is not
                       correct here
E is unpack_sum_1_2 -> pushglobal $unpack_sum_1_2
        where $unpack_sum_1_2 code is
                         - remember that $unpack_sum
                           is called with a function f
                           and a vlue v as arguments
            push 1       - push the value
            eval         - evaluate
            testcons     - test_1_2 actually
                           tests whether the whnf is
                           the first alternative
            jfail L      - jump on fail to L
            push 1       - push the value once again
            sel_sum_1_2  - push the second component
            push 2       - push the value yet again
            sel_sum_1_1  - push the first component
            push 2       - push the function
            mkap
            mkap
            update 3
            pop 2
            unwind
     L:     pushfail     - push the value fail, the
                           previous fail has been consumed
                           by jfail and the surrounding
                           fatbar needs to see a fail.
            update 3
            pop 2
            return

E is if -> pushglobal $if
        where $if code is
           push 0
           eval
           jfalse l1
           push 1
           jump l2
      l1:  push 2
```

```
              l2:  update 4
                   pop 3
                   unwind

       E is [] -> pushglobal $[]
                 where $[] is
                   push 0
                   eval
                   jfail l1
                   push 0
                   jump l2
              l1:  push 1
              l2:  update 3
                   pop 2
                   unwind
```

```
A large example
---------------

 $map f l = if l == [] then []
            else    unpack_1_2 (\x\xs f s : map f xs) l
                 [] error

After lambda lifting

 $xxs f x xs = f x : $map f xs
 $map f l = if l == [] then []
            else    unpack_1_2 ($xxs f) l
                 [] error

 $xxs code:
    push 2                 - xs
    push 1                 - f
    pushglobal $map        -
    mkap
    mkap                   - ($map f x)
    push 2                 - x
    push 2                 - f
    mkap                   - (f x)
    pushglobal $pack_1_2
    mkap
    mkap                   - f x : $map f xs
```

```
   update 4
   pop 3
   unwind

$map:
   push error
   push 2                 - l
   push 2                 - f
   pushglobal $xxs
   mkap                   - ($xxs f)
   pushglobal unpack_1_2  -
   mkap
   mkap
   pushglobal $[]
   mkap
   mkap                   - unpack_1_2 ($xxs f) l
   pushglobal $pack_2_0   - Nil
   pushglobal $pack_2_0
   push 4
   pushglobal $==
   mkap
   mkap                   - l == []
   pushglobal $if
   mkap
   mkap
   mkap                   - if ... [] error
   update 3
   pop 2
   unwind
```